Administrivia 1

ex0 was due an hour ago! Solution posted after class
- Any problems with it (logistics, content, other)?

ex1 out now, due before class Friday

hw0 out tonight
- Logistics and infrastructure - should be quick
- Demos & setup in sections tomorrow - bring a laptop to sections if you can
- Do not be alarmed by email from GitLab when repos are created

Reference system (grading, etc.) is CSE lab/VM Linux
- If you use the CSE VM, be sure it’s the latest one and do ‘sudo dnf upgrade’

hw1 (first big project) out by this weekend
Communications

- Use discussion board when possible
  
  ‣ Contribute & read - help each other out
  
  ‣ **Everyone should must** post a followup to the “welcome” message - get gopost to track new messages for you

- Mail to cse333-staff@cs when needed (not individual staff if possible — and it usually *is* possible)

Anybody still trying to register for the class? Let’s get it done asap so we can set up gitlab and cse accts. for hw0
Today’s agenda

More C details

- functions
- arrays
- refresher on C’s memory model
  - address spaces
  - the stack
  - brief reminder of pointers
Defining a function

```c
returnType name(type name, ..., type name) {
    statements;
}
```

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;

    for (i=1; i<=max; i++) {
        sum += i;
    }
    return sum;
}

sum_fragment.c
Problem: ordering

You shouldn’t call a function that hasn’t been declared yet

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}

int sumTo(int max) {
    int i, sum = 0;

    for (i=1; i<=max; i++) {
        sum += i;
    }
    return sum;
}
```

sum_badorder.c
Problem: ordering

Solution 1: reverse order of definition

```c
#include <stdio.h>

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;
    for (i=1; i<=max; i++) {
        sum += i;
    }
    return sum;
}

int main(int argc, char **argv) {
    printf("sumTo(5) is: %d \n", sumTo(5));
    return 0;
}
```

sum_betterorder.c
Problem: ordering

Solution 2: provide a declaration of the function
- teaches the compiler the argument and return types of the function
- then definitions can be in a logical order, not who-calls-what

```c
#include <stdio.h>

// this function prototype is
// a declaration of sumTo
int sumTo(int);

int main(int argc, char **argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;

    for (i=1; i<=max; i++) {
        sum += i;
    }
    return sum;
}
```

sum_declared.c
Declaration vs Definition

C/C++ make a careful distinction between these

**Definition:** The thing itself

- Code for function; a global variable definition that creates storage
- Must be **exactly one** actual definition of each thing (no dup's)

**Declaration:** Description of a thing, repeated in all files that use it

- Function prototype or external variable declaration
  - Often in header files and incorporated via #include
  - Should also #include declaration in the file with the actual definition to check consistency
- Should appear before first use
Arrays

\[ \text{type name[size];} \]

- example allocates 100 ints’ worth of memory
  - initially, each array element contains garbage data

\[ \text{an array does not know its own size} \]

- \text{sizeof(scores)} is not reliable; only works in some situations
- recent versions of C allow the array size to be an expression
  - But not good practice to put large data in local stack frames (performance)

\[ \text{int } n=100; \]
\[ \text{int } \text{scores}[n]; \quad \text{// OK in C99} \]
Initializing and using arrays

`type name[size] = {value, value, ..., value};`
- allocates an array and fills it with supplied values
- if fewer values are given than the array size, fills rest with 0
- only works for initialization - can’t assign whole array values later

`name[index] = expression;`
- sets the value of an array element

```c
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;  // smash!
```

```c
// 1000 zeroes
int allZeroes[1000] = {0};
```
Multi-dimensional arrays

type name[rows][columns] = {{values}, ..., {values}};

- allocates a 2D array and fills it with predefined values

// a 2 row, 3 column array of doubles
double grid[2][3];

// a 3 row, 5 column array of ints
int matrix[3][5] = {
    {0, 1, 2, 3, 4},
    {0, 2, 4, 6, 8},
    {1, 3, 5, 7, 9}
};
Parameters: reference vs value

Two fundamental parameter-passing schemes

Call-by-value

- Parameter is a local variable initialized when the function is called, but has no connection with the calling argument after that [C: almost everything, Java: everything (primitive types, references values)]

Call-by-reference

- Parameter is an alias for the actual argument supplied in the call (which must be a variable); it is not a separate local variable in the function [C/C++ arrays, C++ references]
Arrays as parameters

It’s tricky to use arrays as parameters

- arrays are effectively passed by reference (not copied)
  - “array promotion” - array name treated as pointer to first element
- arrays do not know their own size

```c
int sumAll(int a[]); // prototype declaration

int main(int argc, char **argv) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers);
    return 0;
}

int sumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < ...???
```
Arrays as parameters

Solution 1: declare the array size in the function

- problem: code isn’t very flexible

```c
int sumAll(int a[5]);

int main(int argc, char **argv) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[5]) {
    int i, sum = 0;
    for (i = 0; i < 5; i++) {
        sum += a[i];
    }
    return sum;
}
Arrays as parameters

Solution 2: pass the size as a parameter

```c
int sumAll(int a[], int size);

int main(int argc, char **argv) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers, 5);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[], int size) {
    int i, sum = 0;

    for (i = 0; i <= size; i++) {  // CAN YOU SPOT THE BUG?
        sum += a[i];
    }
    return sum;
}
```

arraysum.c
Returning an array

Local variables, including arrays, are stack allocated

- they disappear when a function returns

- therefore, local arrays can’t be safely returned from functions (can’t assign/return whole arrays as values)

```c
int *copyarray(int src[], int size) {
    int i, dst[size]; // OK in C99
    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
    return dst; // no -- compiles ok
    // but wrong
}
```

buggy_copyarray.c
Solution: an output parameter

Create the “returned” array in the caller

- pass it as an **output parameter** to copyarray
- works because arrays are effectively passed by reference

```c
void copyarray(int src[], int dst[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

copyarray.c
OS and processes

The OS lets you run multiple applications at once

- an application runs within an OS “process”
- the OS timeslices each CPU between runnable processes
  ‣ happens very fast; ~100 times per second!
Processes and virtual memory

OS gives each process the illusion of its own, private memory

- this is called the process’ address space
- contains the process’ virtual memory, visible only to it
- \(2^{32}\) bytes on 32 bit host
- \(2^{64}\) bytes on 64 bit host
Loading

When the OS loads a program, it:

- creates an address space
- inspects the executable file to see what’s in it
- (lazily) copies regions of the file into the right place in the address space
- does any final linking, relocation, or other needed preparation
The stack

Used to store data associated with function calls

- when you call a function, compiler-inserted code will allocate a stack frame to store:
  - the function call arguments
  - (x86-64 args passed in registers, but copies often saved in frame)
  - the address to return to
  - local variables used by the function
  - a few other pieces of bookkeeping

```c
int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    return x;
}
```

<table>
<thead>
<tr>
<th>offset</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>p2</td>
</tr>
<tr>
<td>24</td>
<td>p1</td>
</tr>
<tr>
<td>16</td>
<td>return address</td>
</tr>
<tr>
<td>12</td>
<td>a[2]</td>
</tr>
<tr>
<td>8</td>
<td>a[1]</td>
</tr>
<tr>
<td>4</td>
<td>a[0]</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>

a stack frame
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ... 
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- main
  - argc, argv, n1
- f
  - p1, p2, x, a
- heap (malloc/free)
- read/write segment
- globals
- read-only segment
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ... 
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc,
         char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...  
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- heap (malloc/free)
- read/write segment
- globals
- read-only segment (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- main
  - argc, argv, n1
- f
  - p1, p2, x, a
- heap (malloc/free)
- read/write segment
- globals
- read-only segment
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...  
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- main
- argc, argv, n1
- heap (malloc/free)
- read/write segment
- globals
- read-only segment
  (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- main
  - argc, argv, n1
- g
  - param
- heap (malloc/free)
- read/write segment
  - globals
- read-only segment
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {  
    int n1 = f(3, -5);  
    n1 = g(n1);  
}

int f(int p1, int p2) {  
    int x;  
    int a[3];  
    ...  
    x = g(a[2]);  
    return x;  
}

int g(int param) {  
    return param * 2;  
}
```

---

<table>
<thead>
<tr>
<th>OS kernel [protected]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td><strong>main</strong></td>
<td>argc, argv, n1</td>
</tr>
<tr>
<td><strong>g</strong></td>
<td>param</td>
</tr>
<tr>
<td><strong>heap (malloc/free)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>read/write segment</strong></td>
<td></td>
</tr>
<tr>
<td><strong>globals</strong></td>
<td></td>
</tr>
<tr>
<td><strong>read-only segment</strong></td>
<td>(main, f, g)</td>
</tr>
</tbody>
</table>
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
The stack in action

```c
int main(int argc,
    char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
Addresses and 

`&foo` produces the virtual address of `foo`

```c
#include <stdio.h>

int foo(int x) {
    return x+1;
}

int main(int argc, char **argv) {
    int x, y;
    int a[2];

    printf("x is at %p\n", &x);
    printf("y is at %p\n", &y);
    printf("a[0] is at %p\n", &a[0]);
    printf("a[1] is at %p\n", &a[1]);
    printf("foo is at %p\n", &foo);
    printf("main is at %p\n", &main);

    return 0;
}
```
Pointers

type *name;            // declare a pointer

type *name = address;  // declare + initialize a pointer

A pointer is a variable that contains a memory address
- it points to somewhere in the process’ virtual address space

```c
int main(int argc, char **argv) {
    int x = 42;
    int *p;       // p is a pointer to an integer
    p = &x;       // p now contains the address of x

    printf("x is %d\n", x);
    printf("&x is %p\n", &x);
    printf("p is %p\n", p);

    return 0;
}
```

A stylistic choice

C gives you flexibility in how you declare pointers

- one way can lead to visual trouble when declaring multiple pointers on a single line

- the other way is what I prefer

```
int* p1; // i prefer
int *p2; // bug?; equivalent to int *p1; int p2;
int* p1, * p2; // correct
```

or

```
int *p1; // correct - better
int *p2; // (int *p1, *p2; is also ok, but less robust)
```
Dereferencing pointers

`*pointer`  // dereference a pointer
`*pointer = value;`  // dereference / assign

dereference: access the memory referred to by a pointer

```
#include <stdio.h>

int main(int argc, char **argv) {
    int x = 42;
    int *p;  // p is a pointer to an integer
    p = &x;  // p now contains the address of x
    printf("x is %d
", x);
    *p = 99;
    printf("x is %d
", x);
    return 0;
}
```
Self exercise #1

Write a function that:

- accepts an array of 32-bit unsigned integers, and a length
- reverses the elements of the array in place
- returns void (nothing)
Self exercise #2

Write a function that:

- accepts a function pointer and an integer as an argument
- invokes the pointed-to function
  - with the integer as its argument
Self exercise #3

Write a function that:

- accepts a string as a parameter
- returns
  ‣ the first whitespace-separated word in the string (as a newly allocated string)
  ‣ and, the size of that word
See you on Friday!